Pokémon is a classic television and video game franchise that takes place in a fantastical world of unique creatures called Pokémon. Pokémon’s gameplay surrounds human trainers who their Pokémon against one another’s Pokémon, using moves and strategies to make the opponent’s Pokémon faint first. The Pokémon game series is a bestseller and began with *Pokémon Blue* and *Pokémon* *Red* versions on the Nintendo Game Boy console in 1996, which are two distinct titles with similar game productions but have a few small feature differences. Early games were released under Nintendo Corporation and run on Game Boy and Game Boy Advanced hardware, which are two of the first Nintendo handheld consoles developed. The player experiences the Pokémon world as a trainer travelling around a region inspired by Japan, defeating other trainers along the way in Pokémon battles, to make his way to the elite ranks of Pokémon trainers.

Our project goal was to create a program that behaves as an autonomous, intelligent Pokémon trainer playing *Pokémon Red* version on a Game Boy console. Our trainer should be able to battle the CPU in the game, and his goal is to win. We started with a Game Boy emulator in place of the Game Boy itself and a Pokémon Red ROM (an emulated image of the actual Game Boy game) to host the game. Game Boy emulators run on PCs and map a subset of keyboard keys to the keys on a Nintendo Game Boy. The specific emulator we used is VisualBoyAdvance Tracer, which is a third-party, open-source emulator with a RAM feature necessary to create our rational agent. VBA Tracer has a“RAM dump” control that flushes the emulator’s working RAM into a readable binary file at every press of the “.” key. The only way for our intelligent program to extract data about the game environment was to read values out of this binary file. It was challenging but interesting to see figure out how old game cartridges like *Pokémon Red* were organized.

The environment of a battle is sequential, static, partially observable in our case, deterministic, discrete, and known. With a completely readable RAM the environment is technically fully observable, but our knowledge of the RAM addresses was limited. We only had a partial map of environment variables to their memory addresses because the only open-source RAM maps found online were incomplete. Therefore in our case it was partially observable. A battle begins as a Pokémon from each of the trainer’s parties (each party can old six Pokémon max) is put on the field. In Pokémon Red there can only be one Pokémon from each party active for battle at a time (but in later generations of the games there are interesting variations). The trainer whose current Pokémon has the higher Speed stat is allowed to make the first move. At each turn a trainer has these possible action choices: command the current Pokémon to use one of its moves, utilize an item in his bag on any of his Pokémon, or switch out the current Pokémon with a different one in the party. Pokémon moves have a variety of consequences – some inflict damage by decrementing the health points of the opponent, others change its own stats or the opposing Pokémon’s stats, or some inflict “statuses” such as poisoning or sleep on its opponent. A battle ends when all the party Pokémon of one trainer have fainted, which means they all have 0 health points left.

We chose to make our trainer a utility-based agent. We decided against a popular but infeasible option of adversarial search, used for many games with known environments, simply because the outcome of an action taken by our agent is based on a probability distribution that depends on dozens of variables. If we had to search through all possible outcomes of all possible actions (there are also a ton of these!) our program would be slow and space-inefficient. We also have seen an implementation online that uses adversarial search to a certain depth (a form of iterative deepening search), which was effective but we wanted to try something different from theirs. There are many elements of randomness for Pokémon games – for example, attack accuracy. Each attack has an accuracy ratio between 0 and 1 – the percentage of times that attack should execute successfully. If an attack’s accuracy is less than 1, occasionally the move will not inflict any damage because it will miss the opponent completely. Therefore we decided against search and settled for a heuristic utility function that we know is not completely optimal but can win a majority of Pokémon battles. Plus, losing a few Pokémon battles isn’t the end of the world, because a trainer can simply “train” his weak Pokémon by battling wild Pokémon. Training with wild Pokémon increases the trainer’s Pokémon’s experience level, which is a highly weighted variable in the calculation of the amount of damage it can do to other Pokémon.

Our utility function was essentially the predicted health point consequence of an action. In simple terms it is the sum of three components we consider important in an outcome.

The predicted damage component is simply a reverse-engineered calculation designed by University of Miami mathematicians (3). They derived a best-fit formula for the amount of damage inflicted by an attacking Pokémon using a certain move on a defending Pokémon. For the predicted damage component, the environment variables our artificial trainer considers are:

* A = attacking Pokémon's experience level. This is the level of the Pokémon who executes the attack. A Pokémon’s level is proportional to the number of Pokémon it has defeated/made faint in its lifetime.
* B = attacking Pokémon's Attack or Special Attack stat. This is essentially the strength of the attacking Pokémon. If the move type is a physical type (normal, rock, flying, ground, etc.) then B = attacking Pokémon’s Attack stat. If the move type is a special type (psychic, water, grass, fire, electric, etc.) then B = Special stat.
* C = selected move’s attack power. This is essentially the strength of the move used in this action.
* D = defending Pokémon's Defense or Special Defense stat. This is the defensive ability of the opponent. If our action is to use a physical move, then D = defending Pokémon’s Defense stat. If our action is a special move, then D = defending Pokémon’s Special stat.
  + For human trainers, this variable is inaccessible but can be heuristically eyeballed or estimated. For example, a rock-type opponent Pokémon tends to have a strong physical Defense stat and a low Special stats. Rock is strong against physical damage but susceptible to water erosion.
* X = same-type attack bonus. X = 1.5 if the attacking Pokémon has the same type as the move it will use. Otherwise, X = 1. This simply gives a bonus to Pokémon who use moves of its type.
* Y = Type modifier, a value calculated by the attacking move’s type and the defending Pokémon’s type.
  + This captures any advantages or disadvantages the attacking move’s type has over the defending Pokémon’s type. For example, the move Water Gun is highly effective on Fire-type Pokémon, giving it a type modifier value of 2. All possible type modifier values are in listed in a chart in source (4), the website of the UMiami mathematicians.
  + We simply stored this chart as a two-dimensional array in our program.
* Z = random int between 217 and 255. This is simply a random seed.

[OTHER ALGO PARTS I’M MISSING – EUGENE]

Our software is structured as a Visual C# console application. We chose C# because we both have Windows machines and needed a reliable API to simulate keyboard presses. Our classes include Program.cs, AITrainer.cs, Control.cs, Utils.cs, and WindowsAPI.cs. The WindowsAPI.cs class was taken from open-source code that basically simulates hardware key presses, because VisualBoyAdvance (the Game Boy emulator) required DirectInput key presses (1). It also gives the VisualBoyAdvance window keyboard focus upon running our program, so that our AI trainer is always focused on the game window. Program.cs contains the main execution method, where we initialize our AITrainer and internal data. AITrainer.cs is just the library of methods that simulate keyboard presses and does not contain any rational calculation, and it invokes methods from WindowsAPI to achieve these simulations. Utils.cs contains useful data structures that represent environment variables in Pokémon Red. For example, it contains an enum of all Pokémon moves, an enum of all Pokémon types, and a map of all variables needed for our utility function to their memory addresses in the Game Boy’s RAM – all hardcoded by us. Finally, Control.cs is where the rational magic happens – here we have “Get” methods for the environment variables, invoke them for calculation of the utility function, search through all values of the utility function, and finally make the optimal move.

There are many ways to evaluate the strength of our artificial trainer. The simplest, most effective measure is putting him in battle as many times as we want, in all sorts of different battle situations with different Pokémon, experience levels, move sets, items, etc. to see the proportion he can win. Unfortunately our artificial trainer cannot play other human trainers and only the Pokémon Red built-in rational agent that controls all trainers besides the human player. However, it is viable and valuable to compare our trainer’s performance to that of a human trainer, or another rational agent, such as the example we found online that inspired us to tackle this project (2).

However, there are many characteristics that make up a qualified Pokémon trainer. In all Pokémon gameplays, the trainer starts his or her journey with one weak starter Pokémon. In Pokémon Red, as he travels through all the cities in the Kanto region, the trainer “catches” wild Pokémon to add to his party, battles Pokémon gym leaders (elite trainers who give you a badge for defeating them, a great honor), and battles other regular trainers. The role-playing journey is long – some Pokémon games take up to 100 hours to complete – and certain adventures and episodes in the storyline comprise of a series of battles with multiple trainers. Move usage is variably limited and finite for each Pokémon, as well, with each move having a certain PP level (the number of times it can be used before healing the Pokémon, which resets the PP). Therefore, it may seem optimal to use your strongest move M1 with PP = 5 that does 200 damage points on an opponent with 100 health points, but it is indeed smarter to use move M2 that inflicts 100 damage points with PP = 15 instead. If the trainer has a long journey before he can reach a Pokémon Center (a Pokémon hospital) then it’s best to conserve the PP of strong attacks for the battles that come towards the journey’s end. So in addition to counting the trainer’s wins, we can assess the shape of his Pokémon after *n* consecutive battles without Pokémon Center or medical access for his poor, tired Pokémon.

This is only one possible extension to our artificial trainer that we didn’t have time to implement. There is so much environment data that we didn’t account for because of sheer storage volume and effort they would require. We would have liked to incorporate machine learning into our trainer. Just as a human player would, our artificial trainer would remember the moves and Pokémon used against him. Our trainer could analyze the opponent’s strategy and try to thwart or counter it. For example, if our opponent repeatedly uses the same move by his current Pokémon that is a physical attack move, it might be smart to have our Pokémon use a move that decreases his Pokémon’s Physical Attack stat.

All in all, this was a fun project and is only the basis of an exploration into the complexities of a Pokémon game. Recent generations of these games are harder to hack and reverse-engineer and feature way more environment variables, so the challenge only gets more difficult. Pokémon Red, however, will remain a classic forever in our hearts, and we can only strive to become or find the ultimate trainer.

Sources:

1. <http://www.ownedcore.com/forums/mmo/warhammer-online/186390-sendinput-example-c.html>
2. <https://www.youtube.com/watch?v=8Yq6lvtgrOo&feature=youtu.be>
3. <https://www.math.miami.edu/~jam/azure/compendium/battdam.htm>
4. <https://www.math.miami.edu/~jam/azure/compendium/typechart.htm>